COMP 364: Computer Tools for Life Sciences

Regular expressions

Christopher J.F. Cameron and Carlos G. Oliver

Key course information

HW4

- due tonight at 11:59:59 pm

HW5

- available now!
- due Thursday, December 7th at 11:59:59 pm

Course evaluations

- available now at the following link:
  - `https://horizon.mcgill.ca/pban1/twbkwbis.P_WWWLogin?ret_code=f`

Outline

Today, we're going to cover **regular expressions** in Python

- what they are
- why they're useful
- how to implement/use them
- etc.

Why not *interpreted vs. compiled languages*?

- we (lightly) covered this topic earlier of the semester
- Carlos will have more to say about it in Friday's lecture
  - *dynamic vs. static typing*

Problem

Let's say you have a large file stored on your laptop

- contains many different email addresses

How would you obtain all email addresses associated with Gmail?

- all Gmail addresses with the letter 'a' in them?
- all Gmail addresses with the substrings 'luv' and 'cats'?
- all Gmail addresses with the substrings 'luv' and 'cats' separated by two characters?
    - luv..cats@gmail.com
    - luvmycats@gmail.com
    - luv48cats@gmail.com

What are regular expressions?

A **regular expression (or regex)** is a sequence of characters

- that helps match or find other strings or sets of strings
- using a specialized syntax held in a pattern

For example:

- `r‘(.*) are (.*?) .*’` is a regex `pattern`
- that would match the following `string`: `"Cats are smarter than dogs"`

Regular expressions are widely used in the world of UNIX

- UNIX is a multitasking, multiuser computer operating systems
- Mac OS is based on UNIX

Why use regex?

Once you learn the syntax of regex

- you'll gain a powerful time-saving tool

It's much faster to write regex patterns

- than to write multiple:
  - conditional statements
  - loops
  - lists
  - variables

Python also makes it very easy to implement regular expressions

- using the `re` module
- API: `https://docs.python.org/3/library/re.html`

Regex in Python and raw stings

When particular characters are used in regular expressions

- they take on a special meaning
- e.g., r'.' means to match any single character except a newline
- does anyone remember what the newline character is?

To avoid any confusion while dealing with regular expressions

- in Python, we use `raw strings` for the `pattern`

To indicate a raw string in python

- prefix the `pattern` string with the 'r' character
- e.g., `r'regex_pattern'`
- e.g., `r'.*'` is different than `'.*'`

Regular Expression Patterns

Except for **control characters**, all characters match themselves

- control characters: $+$? . $*$ $\wedge$ \$ ( ) [ ] { } $\|$ \
- meta characters that give special meaning to the regex

For example, without a control character:

- the pattern `r‘a’` means match the letter ‘a’
- applying the pattern to the string ‘`David likes naan`’
- would return ‘a’ from ‘David’ and two ‘a’s from ‘naan’

With a control character:

- `r‘a{2}’` means match exactly two occurrences of ‘a’
- would return ‘aa’ from ‘naan’

Control characters

1. `r‘∧’` - matches the start of a string
2. `r‘$’` - matches the end of a string
3. `r‘.’`- matches any single character except newline
4. `r‘[...]’` - matches any single character in brackets
   - e.g., `r‘[a-zA-Z]’` matches one occurrence of any ASCII character
5. `r‘[∧...]’` - matches any single character not in brackets
   - similar to Python’s ‘`not`’ in this context

Control characters #2

6. `r'*'` - matches 0 or more occurrences of preceding expression

7. `r'+'` - matches 1 or more occurrence of preceding expression

8. `r'?'` - matches 0 or 1 occurrence of preceding expression

9. `r'n'` - matches exactly *n* occurrences of the preceding expression
   - `r'a{2}'` matches ‘aa’ in ‘naan’

10. `r'a | b'` - matches either ‘a’ or ‘b’

Regex character classes

Character classes (or sets)

- define patterns that match only one out of several characters

For example:

1. `r‘[Pp]ython’` - match ‘Python’ or ‘python’
2. `r‘[aeiou]’` - match any one lowercase vowel
3. `r‘[0-9]’` - match any digit
   - same as `r‘[0123456789]’`
4. `r‘[∧0-9]’` - match anything other than a digit
5. `r‘[a-zA-Z0-9]’` - match any ASCII letter or digit

Quiz

Using the online regex tester at: `https://pythex.org/`

- includes a regex cheatsheet

Provide regex patterns to complete the following:

1. match all occurrences of alphabetical letters
2. match any integer number
3. match any character that precedes the pattern ‘zz’
4. match any string that does not start with ‘p’
5. matches: ‘affgfking’, ‘rafgkahe’, and ‘bafghk’
   but not match: ‘fgok’, ‘a fgk’, and ‘affgm’

You will need to create your own example strings to test for ?’s 1-3

Quiz - solutions

Solutions:

1. `r'[a-zA-Z]+'`
   - `r'[a-zA-Z]'` - matches one occurrence of an ASCII character
   - `r'+'` - matches one or more occurrences of preceding pattern
2. `r'-?[0-9]+'`
   - `r'-?'` - matches zero or one occurrence of '-'
   - `r'[0-9]'` - matches one occurrence of any digit
3. `r'.zz'`
   - `r'.'` - matches one occurrence of any character
   - `r'zz'` - matches one occurrence of 'zz'
4. `r'∧[∧p]+`
   - `r'∧'` - match start of `string`
   - `r'[∧p ]'` - do not match 'p'
5. `r'∧[∧mo ]+$'`
   - `r'$'` - match end of `string`

Regex in Python

The `match()` function

- function attempts to match regex `pattern` at beginning of the `string`
- syntax:
  `re.match(pattern, string, flags=0)`
- parameters:
  1. `pattern` - regular expression to be matched
  2. `string` - string to be searched
  3. `flags` - we'll ignore this optional keyword argument

Regex in Python #2

The `match()` function

- returns a `match` object on success
  - `None` on failure
- to get the matching string
  1. `group(num=0)` - method returns entire match
     - or specific subgroup `num`
  2. `groups()` - returns all matching subgroups in a tuple
     - empty if there weren't any

match() example

```python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line)

if matchObj:
    print("matchObj.group() : ", matchObj.group())
    print("matchObj.group(1) : ", matchObj.group(1))
    print("matchObj.group(2) : ", matchObj.group(2))
else:
    print("No match!!")
```

match() example #2

If the previous code was implemented correctly:

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

By using the ( ) control characters

- specify groups to be matched

Regex in Python #3

The `search()` function

- function searches for first occurrence of `pattern` anywhere within `string`
- syntax:
  `re.search(pattern, string, flags=0)`
- parameters:
  1. `pattern` - regular expression to be matched
  2. `string` - string to be searched
  3. `flags` - we'll ignore this optional keyword argument

Regex in Python #4

The `search()` function

- returns a `match` object on success
  - `None` on failure
- to get the matching string
  1. `group(num=0)` - method returns entire match
     - or specific subgroup `num`
  2. `groups()` - returns all matching subgroups in a tuple
     - empty if there weren't any

search() example

```python
import re

line = "Cats are smarter than dogs"

searchObj = re.search( r'(.*) are (.*?) .*', line)

if searchObj:
    print("searchObj.group() : ", searchObj.group())
    print("searchObj.group(1) : ", searchObj.group(1))
    print("searchObj.group(2) : ", searchObj.group(2))
else:
    print("No match!!")
```

`search()` example #2

If the previous code was implemented correctly:

```
searchObj.group() :  Cats are smarter than dogs
searchObj.group(1) :  Cats
searchObj.group(2) :  smarter
```

Wait, `re.search()` is behaving the same as `re.match()`

- what's the point of having two functions that perform the same operation?

Matching versus searching

Python offers two different operations based on regular expressions

1. `re.match()`
   - checks for a `pattern` match only at the beginning of the `string`
2. `re.search()`
   - checks for a `pattern` match anywhere in the `string`

The second operation is the default of most regex implementations

```
import re

line = "Cats are smarter than dogs"
matchObj = re.match( r'dogs', line)
if matchObj:
    print("match --> matchObj.group() : ",
          matchObj.group())
else:
    print("No match!!")
# prints: No match!!
searchObj = re.search( r'dogs', line)
if searchObj:
    print("search --> searchObj.group() : ",
          searchObj.group())
else:
    print("Nothing found!!")
# prints: search --> matchObj.group() :  dogs
```

Search and Replace

The `sub()` function

- one of the most important `re` methods
- replaces all occurrences of the `pattern` in `string` with `repl`

- syntax:
  `re.sub(pattern, repl, string, max=0)`

- parameters:
  1. `repl` - string to replace `pattern`
  2. `max` - replace all occurrences unless set

- returns a modified string

```python
import re

phone = "2004-959-559 # This is a Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print("Phone Num : ", num)
# prints: Phone Num :  2004-959-559

# Remove anything other than digits
num = re.sub(r'[^0-9]', "", phone)
print("Phone Num : ", num)
# prints: Phone Num :  2004959559
```

Closing comments

We've only covered the basics of **regular expressions**

- there is A LOT more to regex
- for more information:
  `https://docs.python.org/3/howto/regex.html`

Regular expressions are not only limited to Python

- try the BASH command `awk`
  - one of the most powerful command line tools